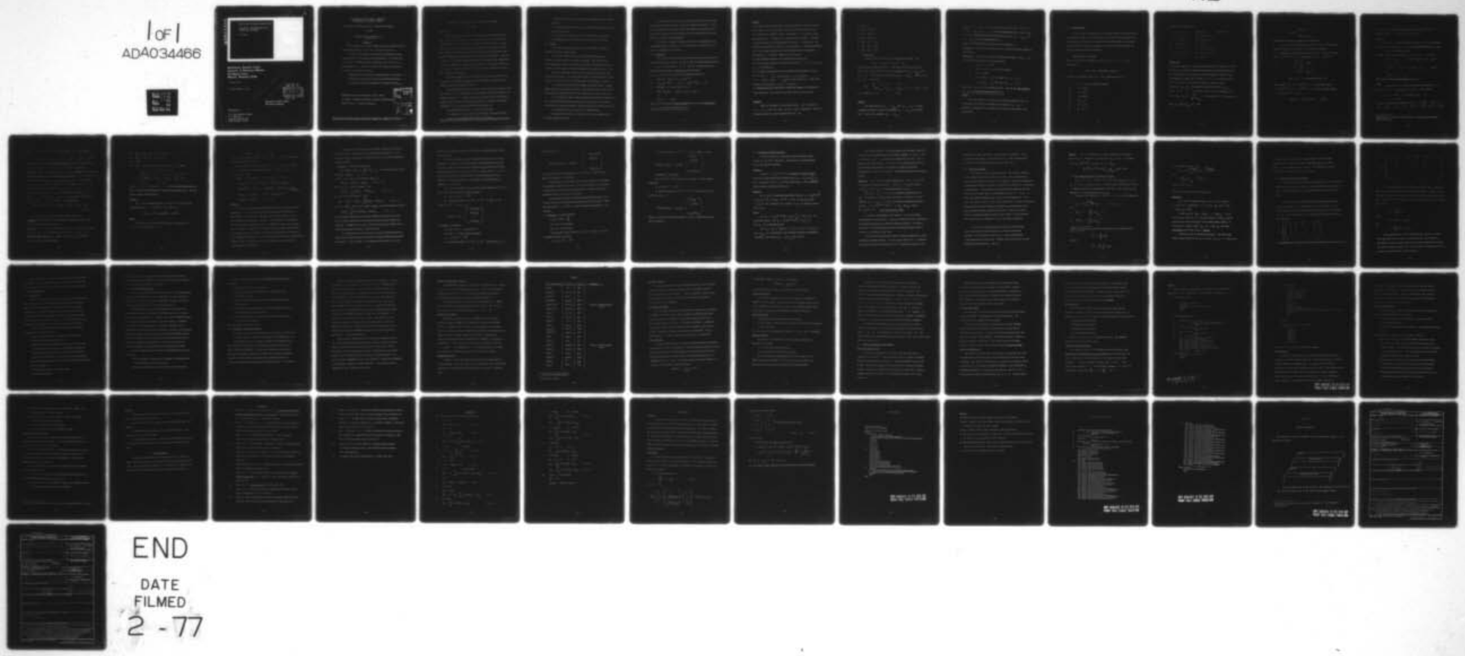
AD-A034 466   WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER   F/G 9/2

AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS.(U)

NOV 76 G KEDEM   DAAG29-75-C-0024

UNCLASSIFIED   MRC-TSR-1697   NL

1 OF 1
ADA034466



END
DATE
FILMED
2 - 77

13

FG

MRC Technical Summary Report #1697

AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS

G. Kedem

Mathematics Research Center
University of Wisconsin–Madison
610 Walnut Street
Madison, Wisconsin 53706

November 1976

Received March 3, 1976

See 1473

Sponsored by

U. S. Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS

G. Kedem

Technical Summary Report #1697
November 1976

ABSTRACT

A method for the automatic differentiation of computer functions (subroutines) written in a high level language is discussed.

A theory is developed to show that most functions that arise in applications can be differentiated automatically. It is shown how one can take a FORTRAN function (subroutine) and, with the aid of a precompiler, obtain a FORTRAN subroutine that computes the original function and its desired derivatives.

Implementation of two types of differentiation is described: in terms of

(1) Automatic Taylor series expansion of FORTRAN programs, and

(2) Automatic Gradient calculation of FORTRAN functions.

AMS(MOS) subject classification: 68.00, 68A15

Key words: Factorable functions, Automatic differentiation

Work Unit No: 7 (Numerical Analysis)

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS

G. Kedem

Introduction

The use of reccurrence relations to compute derivatives is not a new idea. We can trace this idea back as far as 1932 when it was used to compute the Emden functions by means of Taylor series expansion by J. R. Airy (See [1]). This idea has apparently been rediscovered many times. In 1964 R. E. Moore [7] showed how one could automatically get Taylor series expansions of FORTRAN-like expressions to solve initial value problems. See [1,2, 5,7,8,10]. The automatic computation of partial derivatives was implemented in 1967 by A. Reiter and J. Gray [11,14] and later by J. Wertz [12], D. Kuba and L. B. Rall [13], and R. E. Pugh [9]. These are programs known to us but the list is probably not complete.

This paper suggests a way to extend the process to functions that can be written in an algebraic computer language (FORTRAN, ALGOL and so on) namely: piecewise factorable functions.

The theoretical part of this paper was written because we heard too often the statement: "Oh, I believe you can differentiate arbitrary expressions, but FORTRAN programs ?!". We then describe the implementation of two types of differentiation: Taylor series expansion (TAYLOR) and gradient computation (GRADIENT), via the use of the AUGMENT precompiler [3].

The method described has a few distinct advantages:

1) AUGMENT is a highly portable precompiler, therefore with little work it can be implemented on almost any system.

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

2) The packages give a very easy way to interface with any FORTRAN program.

3) Any other type of differentiation can be implemented easily (about two weeks of work).

4) Functions and subroutines can be "differentiated" separately, incorporated into larger systems and used as part of the library.

1.1. Theory.

Let us look at computer programs for the evaluation of numerical functions that arise in applications.

We use the FORTRAN language but the following discussion applies to any other high level algebraic language. We ignore the fact that computers don't really work with real numbers and that library functions like SIN and COS compute only approximations to functions we have in mind.

We look at FORTRAN subroutines and functions that evaluate some mathematical functions. We assume that no I/O is involved and that "random numbers" are not used. All such routines have a few features in common:

1) For every set of values of the formal arguments there is a fixed, finite sequence of instructions executed (provided that these values are within the domain of definition of the function and we use a correct subroutine).

2) If we regard DO loops, logical statements and GOTO statements only as a convenient tool for defining sequences of instructions, we see that all such sequences consist of arithmetic operations and calls to library functions.

3) At each step we use only previously defined values.

4) Most of the functions we compute are piecewise differentiable or actually piecewise analytic.

In order to make the analysis more precise we use an abstract model for algebraic computer languages, namely the factorable functions. But we will try to point out the analogy between the model and computer programs and what the statements about factorable functions mean when we translate them to facts about computer programs.

We will use subscripts to denote sequences and superscripts to denote components of a vector, f_i is the ith element in a sequence and t^j is the jth component of a vector t.

1.2. Definitions.

Let $\mathcal{L}$ be a finite set of real functions of one or more real arguments including the identity function. We call $\mathcal{L}$ the set of basic library functions.

Let f be a map from R^n to R^m. We call f factorable function if and only if there exists a finite sequence of functions $f_1, \ldots, f_k : D \subseteq R^n \rightarrow R$ that satisfy the following conditions:

1) $f_1(x) = x^1,\ f_2(x) = x^2,\ \ldots,\ f_n(x) = x^n$

2) $f_{k-m+1}(x) = f^1(x^1, \ldots, x^n),\ \ldots,\ f_k(x) = f^m(x^1, \ldots, x^n)$

3) for every $i, n < i \leq k$, either $\exists\ g \in \mathcal{L} \ni$

$$f_i(x) = g(f_{j_1}(x), \ldots, f_{j_s}(x)) \qquad j_1, j_2, \ldots, j_s < i,$$

or

$$f(x) \equiv C \qquad C_i \in R,$$

we call such a sequence a basic sequence and we say that the sequence $f_1, \ldots, f_k$ is a basic representation for f.

Remarks.

1) We will not use the properties of the real numbers in most of the discussion to follow, so one could apply the theory to functions over any set.

2) The analogy to FORTRAN is clear. The sequence $\{f_i\}$ that is a basic representation for f corresponds to subroutine that computes f. The first n places correspond to the arguments and the last m to the result. The middle part corresponds to the body of the subroutine. A basic representation for f can actually be written down by following the path of execution (assuming no IF and GOTO statements are involved).

3) Many different sequences can represent the same function.

4) It is easy (but important) to verify that for every $i, n < i \leq k$, the sequence $f_1, \ldots, f_i$ represents f_i.

5) It is always assumed that the compositions are well defined, that is:
If $f = g(f_{j_1}, \ldots, f_{j_s})$ then for every $\underline{x}$ in the domain of f $(f_{j_1}(\underline{x}), f_{j_2}(\underline{x}), \ldots, f_{j_s}(\underline{x}))$ *is in the domain of* g. *Only if this* condition is satisfied, can we call f factorable.

6) *Functions which are identically constant* are thought of as functions of many arguments. The number of arguments depends on the context.

Example:

Here is an example of a factorable function. Let $\mathcal{F}$ be the set $\{+, -, *, /, **, \sin, \cos, \exp\}$ and $f(x) = \cos(x) * \exp(\sin(x)) + \sin(x)**2$. A sequence which is a basic representation for f is:

1) $f_1(x) = x$

2) $f_2(x) = \cos(f_1(x))$

3) $f_3(x) = \sin(f_1(x))$

4) $f_4(x) = \exp(f_3(x))$

5) $f_5(x) = f_3(x) * f_3(x)$

6) $f_6(x) = f_2(x) * f_4(x)$

7) $f_7(x) = f_6(x) + f_5(x)$

1.3. Composition.

Let $q:R^n \to R^k$ and $h:R^k \to R^m$ be factorable functions. Let

$$q_1, q_2, \ldots, q_{L_1+k} \quad \text{and} \quad h_1, h_2, \ldots, h_{L_2}$$

be basic representations for g and h. From these two sequences we construct a third sequence that we call the composition of $q_1, \ldots, q_{L_1+k}$ *with* $h_1, \ldots, h_{L_2}$ ". The sequence is of the form $f_1, f_2, \ldots, f_{L_1+L_2}$ where $f_i = q_i$ for $i = 1, \ldots, L_1 + k$, and

$$f_{L_1+i} = \begin{cases} c & \text{if } h_i \equiv c, \quad c \in R \\ g(f_{L_1+j_1}, f_{L_1+j_2}, \ldots, f_{L_1+j_s}) & \text{if } h_i = g(h_{j_1}, \ldots, h_{j_s}) \end{cases}$$

for $i = k+1, k+2, \ldots, L_2$ $\qquad g \in \mathfrak{L}$

Remark:

The composition of $q_1, \ldots, q_{L_{1+k}}$ with $h_1, \ldots, h_{L_2}$ is simply overlaying the first k terms in the sequence $h_1, \ldots, h_{L_2}$ on top of the last k terms of the sequence $q_1, \ldots, q_{L_{1+k}}$.

Lemma 1: Let q and h be factorable functions $q: R^n \to R^k$; $h: R^k \to R^m$. Let $q_1, \ldots, q_{L_1+k}$ and $h_1, \ldots, h_{L_2}$ be basic representations for q and h, then $f = h(q)$ is a factorable function and the composition of $q_1, \ldots, {}_{L_1+k}$ with $h_1, \ldots, h_{L_2}$ is a basic representation for f.

Proof: The proof follows indirectly from the definitions.

Corollary 1: The set of factorable functions is closed under finite number of substitutions.

Corollary 2: Let $\mathfrak{F}$ be a collection of factorable functions. Let $f_1, \ldots, f_L$ be a sequence of functions of the following form.

1) $f_1 = x^1, \ldots, f_n = x^n$

2) for $n < i \leq L$ f_i is of one of the forms

a) $f_i \equiv$ const.

b) $f_i = g(f_{j_1}, \ldots, f_{j_s})$ for some $q \in \mathfrak{F}$ $j_1, \ldots, j_s < i$,

c) $f_i = q(f_{j_1}, \ldots, f_{j_t})$ for some $q \in \mathfrak{F}$ $j_1, \ldots, j_t < i$.

Then, for $n < j \leq L$, f_j is a factorable function. We say that <u>the sequence $f_1, \ldots, f_j$ is a factorable representation for f_j.</u>

Remark.

The substitution of one sequence into another corresponds to a call to subroutine or function in FORTRAN and the sequence $f_1, \ldots, f_L$ in Corollary 2 corresponds to a subroutine that uses other subroutines in the computations.

1.4. Differentiation:

In this section we show how one can compute total or partial derivatives of factorable functions. We show how one can get from a basic representation for a factorable function to a new sequence of functions which is a factorable representation of the original function and its total (or partial derivatives by means of simple replacement operations.

We start with an example:

Let $\mathfrak{L}$ and f(x) be as in section 1.2; that is, $\mathfrak{L}$ = {+, -, *, /, **, sin, cos, exp} and

$$f(x) = \cos(x) * \exp(\sin(x)) + \sin(x)**2$$

As before the following sequence is a basic representation for f:

1) $f_1 = I$ (I is the identity function)

2) $f_2 = \cos(f_1)$

3) $f_3 = \sin(f_1)$

4) $f_4 = \exp(f_3)$

5) $f_5 = f_3 * f_3$

6) $f_6 = f_2 * f_4$

7) $f_7 = f_6 + f_5$.

We replace this sequence by

1) $\langle F_1(X,X') = X \quad , \quad F_1'(X,X') = X' \rangle \qquad \forall (X,X'') \in R^2$

2) $\langle F_2 = \cos(F_1) \quad , \quad F_2' = -\sin(F_1) * F_1' \rangle$

3) $\langle F_3 = \sin(F_1) \quad , \quad F_3' = \cos(F_1) * F_1' \rangle$

4) $\langle F_4 = \exp(F_3) \quad , \quad F_4' = F_4 * F_3' \rangle$

5) $\langle F_5 = F_3 * F_3 \; . \quad , \quad F_5 = F_3 * F_3' + F_3' * F_3 \rangle$

6) $\langle F_6 = F_2 * F_4 \quad , \quad F_6' = F_2 * F_4' + F_2' * F_4 \rangle$

7) $\langle F_7 = F_6 + F_5 \quad , \quad F_7' = F_6' + F_5' \rangle$.

Please Note:

a) The new sequence is a factorable representation for a function $F : R^2 \rightarrow R^2$

b) There is a "simple" correspondence between the original sequence and the new sequence. Each term in the original sequence of the form $g(f_j)$ was replaced by a term $G(F_j)$ where: $G: R^2 \rightarrow R^2$ is a factorable function, and F_j is the jth term in the new sequence. Similarly terms of the form $g(f_{j_1}, f_{j_2})$ were replaced by terms of the form $G(F_{j_1}, F_{j_2})$ where $G : R^4 \rightarrow R^2$ a factorable function which corresponds to g .

c) Finally, $F(t_0,1) = \begin{pmatrix} f(t_0) \\ \frac{d}{dt} f(t) \Big|_{t=t_0} \end{pmatrix}$ or in general: If h is a function of t,

$h(t_0) = h_0 \, , \; \frac{d}{dt} h(t) \Big|_{t=t_0} = h_0'$ then

$$F(h_0, h'_0) = \begin{pmatrix} f(h(t_0)) \\ \\ \frac{d}{dt} f(h(t))\Big|_{t=t_0} \end{pmatrix} .$$

The following discussion describes the general case.

Let $\mathcal{F}$ be a set of basic library functions. Let T be an operator that maps functions to functions. We assume the following:

1) T is defined for all factorable functions and if $f : D \subseteq R^n \to R^m$ then $T[f] : E \subseteq R^{n \cdot k} \to R^{m \cdot k}$ (k is the degree of T). $E = D \times R^{(k-1)\cdot m}$.

2)
$$T\left[\begin{pmatrix} f^1 \\ \vdots \\ f^m \end{pmatrix}\right] = \begin{pmatrix} T[f^1] \\ \vdots \\ T[f^m] \end{pmatrix}$$

3)
$$T[I_1] = I_k \qquad (I_j \text{ is the identity map in } R^j).$$

4) For every $g \in \mathcal{L}$ $(g : D \subseteq R^s \to R)$ $\exists$ a factorable function $G : E \subseteq R^{s \cdot k} \to R^k$, $(E = D \times R^{(k-1)\cdot s})$, $\ni$ for every factorable function
$\mathbf{f} : \tilde{D} \subseteq R^n \to R^s$, $f(\tilde{D}) \subseteq D$

$$T[g(f^1, \ldots, f^s)] = G(T[f^1], \ldots, T[f^s]) .$$

5) $T[\text{const.}]$ is a constant vector function ($T[\text{const.}] \in R^k$) and $\exists$ a factorable function $C : R \to R^k \ni \forall c \in R \;\; T[c] \equiv C(c)$.

Theorem 1.

Let $\mathfrak{L}$ and T satisfy the above conditions. Let f be a factorable function $f:R^n \to R$, and let $f_1,\ldots,f_L$ be a basic representation for f, then:

$T[f]$ is a factorable function.

moreover if we replace the terms $f_1,\ldots,f_L$ by $F_1,\ldots,F_L$ where

i) for $1 \le i \le n$ $\quad F_i$ is the sequence $x^{i,1},\ldots,x^{i,k}$

ii) for $n < i \le L$

$$F_i = \begin{cases} G_i(F_{j_1},\ldots,F_{j_s}): \text{ if } & f_i = g_i(f_{j_1},\ldots,f_{j_s}) \qquad g_i \in \mathfrak{L}^{\dagger} \\ \\ C(c) \qquad \text{if} & f_i \equiv c \end{cases}$$

then we get a factorable representation for $T[f]$.

Proof:

By induction on the length of the sequence representing f. If $f:R^n \to R$ and $f_1,\ldots,f_L$ represents f and $L = n+1$ then either

a) $\quad f = g(x^{j_1},\ldots,x^{j_s})$ for some $g \in \mathfrak{L}$ $\quad$ or

b) $\quad f = \text{const.}$

in case a we replace the sequence $x^1,x^2,\ldots,x^n,\; g(x^{j_1},\ldots,x^{j_s})$ by $x^{1,1},\ldots,x^{1,k},\ldots,\; x^{n,1},\ldots,x^{n,k},\;\; G(x^{j_1,1},\ldots,x^{j_1,k},\ldots,x^{j_s,1},\ldots,x^{j_s,k})$

†The function G_i is the factorable function corresponding to the basic library function g_i.

where G is the factorable function corresponding to g. By Assumptions 3 and 4

$$T[f] = T[g(x^{j_1},\dots,x^{j_s})] = G(x^{j_1,1},\dots,x^{j_1,k},\dots,x^{j_s,1},\dots,x^{j_s,k})$$

and therefore we have a factorable representation for $T[f]$. In case b if $f \equiv c$ then we replace $x^1,\dots,x^n,c$ by $x^{1,1},\dots,x^{n,k},C(c)$ and again by Assumption 5 $T[c] = C(c)$. Let $f: R^n \to R$, $f_1,\dots,f_L$ represent f, $L = n+j$, and assume the theorem is true for all sequences of length $L = n+i$, $i<j$. $f = f_L(x^1,\dots,x^n)$ is either of the form $g_L(f_{j_1},\dots,f_{j_s})$, $g_L \in \mathfrak{L}$ $j_1,\dots,j_s < L$ or $f_L \equiv \text{const}$. For $f_L \equiv \text{const}$ the same argument as for $L = n+1$ applies. Let $F_1,\dots,F_L$ be the sequence that corresponds to $f_1,\dots,f_L$. $f = f_L = g(f_{j_1},\dots,f_{j_s})$, so by construction, $F_L = G(F_{j_1},\dots,F_{j_s})$. By the induction hypothesis, $F_{j_1} = T[f_{j_1}],\dots,F_{j_s} = T[f_{j_s}]$ and F_{j_i}, $1 \le i \le s$, are factorable, therefore: By Lemma 1, $G(F_{j_1},\dots,F_{j_s})$ is factorable. By Assumption 4,

$$T[f_L] = T[g(f_{j_1},\dots,f_{j_s})] = G(T[f_{j_1}],\dots,T[f_{j_s}]) = G(F_{j_1},\dots,F_{j_s})$$

$\because$ Q. E. D.

As an immediate consequence of this theorem we have:

Corollary 3: The theorem is true for factorable functions from R^n to R^m.

We also have the following extension.

Theorem 2: Let T and $\mathfrak{L}$ satisfy the above. Let $\mathfrak{F}$ be a set of factorable functions. Let f be a factorable function $f: R^n \to R$, and $f_1,\dots,f_L$ a factorable representation for f, that is: $f_1 = x^1,\dots,f_n = x^n$ and for $n < i \le L$ f_i is of one of the forms

a) $f_i = q(f_{j_1}, \dots, f_{j_s}), \quad j_1, \dots, j_s < i, \quad g \in \mathfrak{L}$

b) $f_i \;\; q(f_{j_1}, \dots, f_{j_t}), \quad j_1, \dots, j_t < i, \quad q \in \mathfrak{F}$

c) $f_i \equiv$ const.

If we replace this sequence by the sequence $F_1, \dots, F_L$ where,

$$F \;\; = x \;\; ^{,1}, \dots, \;\; ^{,k} \text{ and for } n < i \leq L$$

$$F_i = \begin{cases} G(F_{j_1}, \dots, F_{j_s}) & \text{if} \quad f_i = g(f_{j_1}, \dots, f_{j_s}), \quad g \in \mathfrak{L} \\ T[q](F_{j_1}, \dots, F_{j_t}) & \text{if} \quad f_i = q(f_{j_1}, \dots, f_{j_t}), \quad q \in \mathfrak{F} \\ C(c) & \text{if} \quad f_i \equiv c \end{cases}$$

then, for $n < \ell \leq L$, the sequence $F_1, \dots, F_\ell$ is a factorable representation for $T[f_\ell]$. The proof of this theorem is the same as the previous one. We only have to use the following lemma.

Lemma 3:

Let f be a factorable function $f : D \subseteq R^n \to R$. Then, for every factorable function $\hat{f} : \hat{D} \subseteq R^m \to R^n$ with $\hat{f}(\hat{D}) \subseteq D$,

$$T[f(\hat{f}^1, \dots, \hat{f}^n)] = T[f](T[\hat{f}^1], \dots, T[\hat{f}^n]) .$$

Proof:

By induction on L, the length of the sequence representing f. If $L = n + 1$ then either $f \equiv c$ or

$$f = g(x^{j_1}, \dots, x^{j_s}) \qquad g \in \mathcal{S}.$$

If $f \equiv c$ there is nothing to prove and if $f = g(x^{j_1}, \dots, x^{j_s})$ then the lemma is true by Assumptions 3 and 4.

Assume the lemma is true for all $L = n+j \quad j < i$ and $L = n+i$, then if $f \equiv c$, again there is nothing to prove and if $f = g(f_{j_1}, \dots, f_{j_s}) \quad j_1, \dots, j_s < L \quad g \in \mathcal{S}$ then:

$$T[f(\hat{f}^1, \dots, \hat{f}^n)] = T[g(f_{j_1}(\hat{f}^1, \dots, \hat{f}^n), \dots, f_{j_s}(\hat{f}^1, \dots, \hat{f}^n))]$$

$$= G(T[f_{j_1}(\hat{f}^1, \dots, \hat{f}^n)], \dots, T[f_{j_s}(\hat{f}^1, \dots, \hat{f}^n)]) \quad \text{by Assumption 4}$$

$$= G(T[f_{j_1}](T[\hat{f}^1], \dots, T[\hat{f}^n]), \dots, T[f_{j_s}](T[\hat{f}^1], \dots, T[\hat{f}^n])) \quad \text{by induction hypotheses}$$

$$= T[g(f_{j_1}, \dots, f_{j_s})](T[\hat{f}^1], \dots, T[\hat{f}^n]) \quad \text{by Assumption 4}$$

$$= T[f](T[\hat{f}^1], \dots, T[\hat{f}^n]) \quad \Longrightarrow \quad \text{Q. E. D.}$$

<u>Remarks:</u>

1) Take $\mathcal{S}$ to be the set of standard Fortran functions and the arithmetic operations. Let T be an operator we wish to implement and assume that Assumptions 1-5 are satisfied. Theorem 1 then says the following: Let F be a subroutine that computes a function f (For the moment we assume that no IF or GOTO statements are used). Suppose we replace each variable by a K vector (L vector by a K × L-array and so on), replace each constant by a constant vector, and replace each arithmetic operation and function call by a corresponding subroutine. Then the result would be a Fortran subroutine that computes T[f]. The replacement rules are simple and can be carried out mechanically by a precompiler.

2) Theorem 2 says that if we are using other subroutines or functions in the course of the computation we can transform them separately. In the replacement process we replace the original call by a call to the corresponding transformed routine.

1.5 Example 1. Taylor Series Expansion

Let $[X(t_0)]_i$ denote $\frac{1}{i!} \frac{d^i}{dt^i} X(t)\Big|_{t = t_0}$. Assume we know the values of $[X(t_0)]_j$, $[Y(t_0)]_j$, $j = 0,\ldots,k$.

i) If $Z = X \pm Y$, then we can compute $[Z(t_0)]_j$ by

$$[Z(t_0)]_j = [X(t_0)]_j \pm [Y(t_0)]_j , \qquad j = 0,1,\ldots,k.$$

ii) If $Z = X * Y$, then by Leibnitz' rule

$$[Z(t_0)]_j = \sum_{s=0}^{j} [X(t_0)]_s \cdot [Y(t_0)]_{j-s} , \qquad j = 0,\ldots,k .$$

iii) If $Z = X/Y$ and $Y(t_0) \neq 0$, then

$$[Z]_j = 1/Y(t_0) \cdot \{[X(t_0)]_j - \sum_{s=0}^{j-1} [Z(t_0)]_s \cdot [Y(t_0)]_{j-s}\} \qquad j = 1,2,\ldots,k .$$

iv) If $Z = EXP(X)$, then we can compute $[Z(t_0)]_j$ using the recurrsion relations

$$[Z(t_0)]_j = \sum_{s=0}^{j-1} ((j-s)/j)[Z(t_0)]_s \cdot [X(t_0)]_{j-s} \qquad j = 1,2,\ldots,k .$$

It is well known that there are recurrsion relations that enable one to compute successive derivatives of functions that satisfy rational differential equations. In Appendix A we give a list of such recurrsion relations for the most common special functions. (See also [1,7]).

As a matter of fact, the discussions in [1] and [7] show that one can write such recurrsion relations for functions satisfying differential equations of the form $Y' = f(t,Y)$ where f is a factorable function and $\mathfrak{L}$ is a set containing the arithmetic operations and functions satisfying rational differential equations.

Let $\mathcal{F}$ be a set that consists of the identity function, the arithmetic operations, functions that satisfy first order rational differential equations (like exp and log), pairs of functions satisfying second order rational differential equations (like sin and cos), and so on. Choose the set $\mathcal{F}$ in such a way that all recurrsion relations between successive derivatives can be expressed as factorable functions. For example: the set of functions in Appendix A is such a set.

Let $k \geq 1$ be an integer. For each basic library function $g : D \subset R^s \rightarrow R$ take G to be the factorable function satisfying:

a) $G : E = D \times R^{s \cdot k} \rightarrow R^{k+1}$

b) for every function $X : R \rightarrow R^s$, $X \in C^k$, if $X(t_0) \in D$ and $[X(t_0)]_i = X_i \in R^s$, $i = 0, 1, \ldots, k$

$$G(X_0, X_1, \ldots, X_k) = \begin{pmatrix} [g(X(t_0))]_0 \\ [g(X(t_0))]_1 \\ \vdots \\ [g(X(t_0))]_k \end{pmatrix}$$

We define T_k as follows:

Let $f : W \subseteq R^s \rightarrow R$ be of class C^k.

Let $T[f] = F$ be the function satisfying:

a) $F : W \times R^{s \cdot k} \rightarrow R^{k+1}$

b) For every function $X : R \rightarrow R^s$; $X \in C^k$, and for every $t_0 \in R$

such that $X(t_0) \in W$

$$F([X(t_0)]_0, [X(t_0)]_1, \ldots, [X(t_0)]_k) = \begin{pmatrix} [f(X(t_0))]_0 \\ [f(X(t_0))]_1 \\ \cdot \\ \cdot \\ \cdot \\ [f(X(t_0))]_k \end{pmatrix} .$$

It is not hard to see that the operator T_k and the set $\mathfrak{L}$ defined above satisfy assumptions 1-5 of Theorem 1.

Note that the set $\mathfrak{L}$ contains only analytic functions and therefore the factorable functions are analytic. In the next section we will show that it is possible to include in the basic library set, functions that are only piecewise analytic (like max, min and abs).

Also note that in order to compute high order derivatives of a function one has to use all the lower order derivatives of that function. Therefore the operator $T : f \to f^{(k)}$, $k > 1$ does not satisfy assumptions 1-5 of Theorem 1.

1.6 <u>Example 2:</u> Partial derivatives

Let $f_{(j)}$ denote $\frac{\partial f}{\partial x_j}$.

Let $\mathfrak{L}$ be as in Example 1.

Let $k \geq 1$ be an integer.

For each basic library function $g : D \subseteq R^s \to R$ take G to be the factorable function satisfying:

a) $G : D \times R^{s \cdot k} \to R^{k+1}$

b) For every function $h: R^k \to R^s$, $h \in C^1$ if $h(X_0) \in D$ then

$$G(h(X_0), h_{(1)}(X_0), \ldots, h_{(k)}(X_0)) = \begin{pmatrix} g(h(X_0)) \\ g_{(1)}(h(X_0)) \\ \vdots \\ g_{(k)}(h(X_0)) \end{pmatrix}$$

We define T_k as follows:

Let $f: W \subseteq R^s \to R$ be of class C^1, let $T_k[f] = F$ be the function satisfying:

a) $F : W \times R^{s \cdot k} \to R^{k+1}$

b) For every function $h: R^k \to R^s$ $h \in C^1$, and for every $X_0 \in R^k$ such that $h(X_0) \in W$

$$F(h(X_0), h_{(1)}(X_0), \ldots, h_{(k)}(X_0)) = \begin{pmatrix} f(h(X_0)) \\ f_{(1)}(h(X_0)) \\ \vdots \\ f_{(k)}(h(X_0)) \end{pmatrix}$$

Again it is not hard to see that the above $\mathfrak{F}$ and T_k satisfy the assumptions of Theorem 1.

1.7. Piecewise factorable functions:

So far the model does not describe computer programs which include IF and GOTO statements. The extension is straightforward. We use the following definition:

Definition:

A function $f : D \subseteq R^n \to R^m$ is a piecewise factorable function if and only if there exists finite number of sets $U_1, \ldots, U_k \ni D \subseteq \bigcup_{j=1}^{k} U_j$ and f restricted to each U_j is a factorable function. A very useful fact about piecewise factorable functions is:

Lemma 4:

Let q and h be piecewise factorable functions. $q : D_1 \subseteq R^n \to R^k$, $h : D_2 \subseteq R^k \to R^m$ if $q(D_1) = \{z \in R^k \mid z = q(x),\ x \in D_1\} \subseteq D_2$ then $f : D_1 \subseteq R^n \to R^m$ defined by $f(x) = h(q(x))$, $x \in D_1$ is a piecewise factorable function.

Proof:

Let $U_1, \ldots, U_s \subseteq R^n$ be sets $\ni D_1 \subseteq \bigcup_{i=1}^{s} U_i$ and $q|_{U_i}$ is a factorable function. Let $V_1, \ldots, V_t \subseteq R^k$ be sets $\ni D_2 \bigcup_{j=1}^{t} V_j$ and $h|_{V_j}$ is a factorable function.

Let $W_{i,j} = \{\underline{x} \in U_i \mid q(\underline{x}) \in V_j\}$ $1 \leq i \leq s$, $1 \leq j \leq t$. If $W_{i,j} \neq \emptyset$ then $f|_{W_{i,j}}$ is a composition of two factorable functions and therefore factorable. Since $q(D_1) \subseteq D_2$ $D_1 \subseteq \bigcup_{i,j} W_{i,j}$. Q.E.D.

Since locally (that is: on the appropriate sets) piecewise factorable functions are factorable, the previous discussion applies. If $\mathfrak{L}$ and T are as in section 1.4 and f is a piecewise factorable function then, T[f] is a piecewise factorable function. If we carry out the replacement process for the factorable representation of each piece, we will get factorable representation for T[f] on each piece. We therefore arrive at the following theorem.

Theorem 3: Let $\mathfrak{L}$ and T satisfy assumptions 1-5. Let $f : D \subseteq R^n \to R^m$ be a piecewise factorable function. Let $U_1, \ldots, U_s \subseteq R^n$ be sets $\ni D \subseteq \bigcup_{i=1}^{s} U_i$ and $f|_{U_i}$ is a factorable function, $1 \leq i \leq s$.

Let $f_{i,1}, \ldots, f_{i,L_i}$, $1 \leq i \leq s$ be factorable representations for $f|_{U_i}$. If we carry out the replacement process as in Theorem 2 for each sequence $f_{i,1}, \ldots, f_{i,L_i}$ $i \leq i \leq s$, then we get a piecewise factorable function , $\hat{F}: D \times R^{(k-1)\cdot n} \to R^{m\cdot k}$. $\ni \hat{F}|_{\hat{U}_i} = T[f|_{U_i}]$ where $\hat{U}_i = U_i \times R^{n\cdot(k-1)}$. We call this function T[f].

In practice, as Examples 1 and 2 show, $\mathfrak{L}$ is a set of analytic and piecewise analytic functions. Also, most functions we use in applications are piecewise factorable and therefore piecewise analytic. Of course, if one wants to talk about piecewise analytic functions, the pieces (that is the sets U_i) should be "nice" sets.

Most computer programs that arise in numerical applications compute piecewise factorable functions. We can always make $T[f]^1 = f$. Therefore if we leave the decision statement and GO TO statement unaltered by the replacement process, we will not change the path of execution. Since locally this path defines a factorable function f, after the replacement we will get a factorable function which is (locally) $T[f]$.

1.8. Iterative procedure:

Many functions are computed iteratively. The number of iterations in the computer program must be finite of course. This number however can change according to the values of the arguments. The usual arrangement in iterative procedure is as follows: One prescribes a tolerance ϵ and sometimes an initial guess. The program then proceeds with the iterations until the change in the function value, or the estimated error is $\leq \epsilon$. Once ϵ is fixed, the number of iterations as a function of the arguments is, in most cases, piecewise constant. Since interative procedure can differ considerably, we cannot say what are the precise conditions that make functions that are computed iteratively, piecewise factorable. However by careful study of a particular problem at hand, in many cases, one can show that the function actually computed is in fact piecewise factorable.

In such a case if one computes derivatives of that function, one actually computes the derivatives of a piecewise factorable function. These derivatives might not be a good approximation to the derivatives we had in mind. However many times one can use the following classical theorem (see [15]).

Theorem: If f_j is a sequence of analytic functions in the complex plane and $f_i \to f$ uniformly on a closed disc $D(x_0, r)$ then f is analytic in the disc, $f_j^{(k)} \to f^{(k)}$, uniformly and $\forall z \in D(x_0, r)$

$$\frac{1}{k!} |f_j^{(k)}(z) - f^{(k)}(z)| \leq \frac{1}{r^k} \sup_{w \in D(x_0,r)} |f_j(w) - f(w)| .$$

1.9 Taylor Series Expansion of Implicit Functions.

Let $f \in C^k$, $(k > 1)$, $f: R \times R^n \to R^n$ assume that $f(t_0, x_0) = 0$, $t_0 \in R$, $x_0 \in R^n$ and $\Gamma = f_x^{-1}(t_0, x_0)$ exists. Then the system of equations $f(t, x(t)) = 0$ defines implicitly a unique function $x: R \to R^n$, $x \in C^k \ni x(t_0) = x_0$ and $f(t, x(t)) \equiv 0$ in the neighborhood of (t_0, x_0).

Moreover, from Theorem 20.3 in [6, Ch. 5, p. 329-332] it follows that

if

$$g_i(t) = f(t, \sum_{j=0}^{i-1} \frac{x^{(j)}(t_0)}{j!}(t-t_0)^j) \qquad i = 1, 2, \ldots, k$$

then

i) for $m < i$ $$\frac{d^m}{dt^m} g_i(t)\Big|_{t_0} = 0$$

ii) $$x^{(i)}(t_0) = -\Gamma \frac{d^i}{dt^i} g_i(t)\Big|_{t_0}^{\dagger} .$$

---

†There is an error in the statement of the theorem in [6] (Theorem 20.3, Ch. 5, p. 329).

$$x^{(i)} = -\Gamma \frac{d^i}{dt^i} (g_i(t))$$

and not

$$x^{(i)} = -\frac{1}{i!}\Gamma \frac{d^i}{dt^i} (g_i(t)) .$$

As a corollary we have: Let

$$\hat{g}_i(t) = f(t, \sum_{j=0}^{i-1} \frac{x^{(j)}(t_0)}{j!}(t-t_0)^j + \frac{\hat{x}\,(t-t_0)^i}{i!})$$

then

$$-\Gamma \frac{d^i}{dt^i} \hat{g}_i(t) \Big|_{t=t_0} = x^{(i)}(t_0) - \hat{x}$$

for any vector $\hat{x} \in R^n$.

The above discussion shows the following:

Theorem 4:

Let f be a factorable function (with $\mathcal{L}$ as in Example 1). $f : R \times R^n \to R^n$. Assume that: $f \in C^s$ $(s \geq 2)$, $f(t_0, x_0) = 0$ and $\Gamma = f_x^{-1}(t_0, x_0)$ exists.

Assume we know $x(t_0)$, $[x(t_0)]_1, \ldots, [x(t_0)]_{i-1}$ $(i \leq s)$. If we take any vector in R^n as an initial guess for $[x(t_0)]_i$ and we get the Taylor series expansion of the modified Newton method (Γ considered as constant matrix) $x_{k+1} = x_k - \Gamma f(t_0, x_k)$ then after one iteration we will get $[x(t_0)]_i$ exactly:

Therefore we use the following procedure: We start by the regular Newton method to find x_0 such that $f(t_0, x_0) = 0$. Then we set

$\Gamma = f_x^{-1}(t_0, x_0)$ and get the Taylor series expansion of the modified Newton method. In the ith step we compute derivatives up to order i, taking 0 as the initial guess for $[x(t_0)]_i$. We continue the same way until we get derivatives up to the desired order.

1.10 Computation of Sparse Gradients

a) Band gradients: Many times the gradient matrix is in band form (for example in the numerical solution of a two-point boundary value problem). In such a case the gradient can be computed with great saving in time and space.

Let $f: R^n \to R^n$ be a factorable function (piecewise factorable). and assume that, for all j, f^j depends only on $\{x^i \mid$ where $|i-j| \le k\}$. Assume that we want to compute the partial derivatives of f with respect to $x \in R^n$. Instead of reserving $n+1$ words for each variable and starting with the matrix:

$$n+1 \left\{ \begin{pmatrix} X^1 & X^2 & X^3 & \cdots & X^n \\ 1 & 0 & & & 0 \\ 0 & 1 & & & 0 \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & 0 \\ \vdots & \vdots & & & 1 \end{pmatrix} \right.$$

One needs to reserve only $2k+2$ words for each variable and start with the matrix

$$2k+2\left\{\begin{pmatrix} x^1 & x^2 & \cdots & x^{2k+1} & x^{2k+2} & x^{2k+3} & \cdots & x^n \\ 1 & 0 & & \vdots & 1 & 0 & & \\ 0 & 1 & \ddots & \vdots & 0 & 1 & & \\ \vdots & 0 & \ddots & & \vdots & 0 & \ddots & \\ & \vdots & & \ddots & \vdots & \vdots & & \\ & & & 0 & & & & \\ 0 & 0 & \cdots & 1 & 0 & 0 & & \end{pmatrix}\right.$$

b) The general case: Many times the gradient matrix is sparse but does not have a structure that is easy to take advantage of. However it is not necessary to know in advance which of the entries of the matrix are identically zero. One can carry this information with the computation and use the following obvious fact:

If

$$J_t = \{i \mid \frac{\partial g_t}{\partial x_i} \not\equiv 0\}$$

and if

$$h(x) = f(g_1(x), \ldots, g_s(x))$$

then

$$J = \{i \mid \frac{\partial h}{\partial x_i} \not\equiv 0\} \subseteq \bigcup_{t=1}^{s} J_t .$$

In the implementation, each variable will be a pointer to a vector which will keep a list of nonzero partial derivatives. Each subroutine that replaces a call to a basic library function will compute function values and nonzero partial derivatives only. The subroutine will create a list of the nonzero partial derivatives of the composition.

The implementation of automatic computation of partial derivatives of FORTRAN functions (GRADIENT) described in this report does not use the above method. We plan to implement this method in the near future.

2. The Implementation.

2.1. Introduction.

In earlier sections it was pointed out that most computer functions and subroutines used in numerical computations compute (represent) piecewise factorable functions. It was shown that every sequence representing a piecewise factorable function can be transformed into another sequence which represents the original piecewise factorable function and its total or partial derivatives. The translation process is merely a replacement process and can be carried out mechanically.

In order to implement such a replacement process one needs a processor that will do the following:

a) Break the subprogram into a sequence of one step terms.

b) Replace each term by a body of code.

c) Expand each program variable into a vector. The size of that vector will depend on the order of the operator implemented.

d) The processor should leave the control structure unaltered, that is: Do loops and IF statements should be left unaltered.

There are three principal ways to implement the replacement process.

a) Macro expansion.

b) Replacing each term by a subroutine call.

c) Using an interpreter.

We chose to use the second method mainly because we could use an existing precompiler: AUGMENT. We also feel that the second method is the most powerful and flexible of the three.

2.2. The AUGMENT Precompiler.

Since we are using AUGMENT as the main tool for the implementation of automatic differentiation, we find it appropriate to give a very short description of the function and use of AUGMENT. However, in order to understand fully how to use it, the user should read [3].

The AUGMENT precompiler was designed to simplify the use of nonstandard data types in FORTRAN. AUGMENT enables one to define new data types and operations. It enables one to write FORTRAN programs using these new data types as though they were standard. AUGMENT input consists of programs written in "extended" FORTRAN, that is; FORTRAN programs using nonstandard data types, operators, and functions. AUGMENT translates the input programs into standard FORTRAN programs with the nonstandard constructs translated into subroutine and function calls. The supporting package (that is, the above subroutines and functions) implement the operations with the nonstandard data types.

In order to implement a new data type with AUGMENT one has to do two things

1) write a package of subroutines to implement the operations and functions defined on the new data type,
2) write a description deck which describes the new data type.

In the description deck one provides AUGMENT with the following information:

a) The name(s) of the new data type(s).

b) The number and type of computer words to reserve for each nonstandard variable.

c) The set of operations and "standard" functions defined on the new data type.

d) The names and calling sequences of the subroutines and functions that implement the above operations.

e) The relations between the new data types and other data types (standard and nonstandard).

For more detailed information see [3].

2.3. GRADIENT and TAYLOR Packages.

This report describes the implementation of two types of differentiation:

1) TAYLOR: Automatic Taylor series expansion of FORTRAN functions.

2) GRADIENT: Automatic gradient computation of FORTRAN functions.

The automatic differentiation is implemented by providing two new data types: TAYLOR and GRADIENT. The operations defined on the new data types are the arithmetic operations and almost all the standard FORTRAN functions. Each subroutine that implements a nonstandard operation computes (represents) the corresponding factorable function G of Theorem 1.

Suppose one wants to compute the gradient of a function f, which is a function of n variables. One has only to write a FORTRAN function (subroutine) that computes f. In the function or subroutine code, one declares the arguments and other variables (including the function itself) as type GRADIENT. Then one submits this function with the description deck (which is part of the package) to AUGMENT as data. The output from AUGMENT will be the desired subroutine. AUGMENT will translate the function into a FORTRAN subroutine written in ANSI standard FORTRAN, declaring each GRADIENT variable as a REAL vector of dimension $n + 1$ (and each k vector as an $(n + 1) \times k$ REAL array and so on). Each arithmetic operation or function call will be translated to a call to the appropriate subroutine. The translated subroutine together with the subroutines provided by the GRADIENT package will compute the gradient of the function f at any desired point.

Below we give a detailed description of the two packages and their use. The complete listing of the supporting packages and the description decks is given on a microfiche card at the back of this report. Most of the details of the two packages are the same: Anything that is said below applies to both packages, unless the contrary is explicitly stated. We will use the term VARIABLE for either type GRADIENT or TAYLOR and CONSTANT for types REAL, INTEGER or DOUBLE PRECISION. The relations between VARIABLE and type COMPLEX are undefined.

TAYLOR and GRADIENT Variables:

Each GRADIENT variable is a REAL vector of dimension $N + 1$ where N is the number of the independent arguments. The first word holds the variable value and the $(I + 1)$th word holds the partial derivative of that variable with respect to the Ith independent argument.

Each TAYLOR variable is a Real vector of dimension $N + 1$ where N is the highest normalized derivative to be computed. The $(I + 1)$th place holds the Ith normalized derivative, $I = 0, 1, \ldots, N$.

Arithmetic Operations:

All arithmetic operations between VARIABLEs and all arithmetic operations between VARIABLE and CONSTANT are legal except INTEGER raised to a VARIABLE power. Since the recurrsion relations that replace arithmetic operations between CONSTANT and VARIABLE are simpler than the general relations, separate routines are provided to implement the arithmetic operations between CONSTANTs and VARIABLEs. Conversion of CONSTANT to a vector format is done if there is a statement of the form $V = c$ where V is a VARIABLE and c is a REAL expression, or if there is a reference to a conversion function (see Conversion routines).

Standard Functions:

In Table 1, we list the standard functions that are implemented in the two packages. One can easily add other functions to that list by adding their names to the description deck and writing subroutines to implement them.

Table 1.

Function reference	Function	Suffix	Comments
ABS(x)	$\|X\|$	ABS	
ACOS(x)†	$\cos^{-1}(x)$	ACS	
ALOG(x)‡	$\ell n(x)$	LN	
ALOG10(x)	$\log_{10}(x)$	LOG	
AMAX1(x,y)‡	max(x,y)	MAX	function of two arguments only
AMIN1(x,y)‡	min(x,y)	MIN	"
ATAN(x)	$\tan^{-1}(x)$	ATN	
ASIN(x)†	$\sin^{-1}(x)$	ASN	
CBRT(x)†	$X^{1/3}$	CBR	
COS(x)	cos(x)	COS	
COSH(x)†	cosh(x)	CSH	
COTAN(x)†	cotan(x)	CTN	
EXP(x)	exp(x)	EXP	
LOG(x)‡	$\ell n(x)$	LN	
MAX(x,y)	max(x,y)	MAX	function of two arguments only
MIN(x,y)	min(x,y)	MIN	"
SIN(x)	sin(x)	SIN	
SINH(x)†	sinh(x)	SNH	
SQRT(x)	$X^{\frac{1}{2}}$	SQR	
TAN(x)†	tan(x)	TAN	

† Not an ANSI standard function.

‡ See automatic typing.

Automatic Typing:

This package provides the same feature that exists in many FORTRAN compilers, automatic typing of functions; that is, the type of function used is determined by its arguments. Thus, for example, we have defined LOG and ALOG to be names of the function which takes the logarithm of an argument of type VARIABLE.

Conversion Functions:

There are three subroutines that implement conversion from CONSTANT to VARIABLE, one each for types REAL, INTEGER and DOUBLE PRECISION. These routines can be referenced in the original program by the use of the conversion function: CTTYL(·) in TAYLOR and CTGRD(·) in GRADIENT. The function accepts all three standard types as arguments (see automatic typing). Automatic conversion is invoked only for type REAL. That is: the statement V = const is legal only if the const. is of type REAL.

Norm Function:

It is sometimes convenient to test the distance between two VARIABLES (for example in a test of convergence). Since the relational operators compare only the first words of the VARIABLES they cannot be used for that purpose. The packages provide a function NORM that computes the distance of a VARIABLE from the 0 vector. In TAYLOR package, the function NORM is a function of two arguments, TAYLOR and REAL

$$\mathrm{NORM}(v, t) = \max_{0 \leq i \leq N} |[v]_i| t^i .$$

In GRADIENT, NORM is a function of one argument

$$\text{NORM}(V) = \max_{0 \le i \le N} (|V_i|) .$$

In both packages the function is implemented as REAL function.

Logical Statements:

The relational operators can be used to compare two VARIABLEs, or VARIABLE with type REAL. The comparison is done between the first words of the VARIABLEs or between the first word of the VARIABLE and type REAL. The comparison operators are implemented as LOGICAL functions.

Other Subroutines:

The packages provide two additional subroutines:

1) Error handling subroutine (see our later discussion of Error handling).

2) Copy subroutine.

The copy subroutine implements the statement A = B, A and B VARIABLEs.

Subroutine Names:

The names of all subroutines in both supporting packages are composed of two parts:

i) The first three letters (the prefix).

ii) The last three or two letters (the suffix).

All the routines in each package have a common prefix: TYL in TAYLOR and GRD in GRADIENT. In order to avoid name conflicts, the user should avoid using names starting with the above prefixes.

The suffix of a subroutine's name depends on the function or operation the supporting routine implements. In Table 1, we give the suffixes of the routines implementing the "Standard" functions. The suffix of a routine implementing arithmetic operations is given systematically. The first letter describes the operator. A for +, S for -, M for *, D for /, and E for **. The next two letters describe the operands: first the left operand and then the right one. The letter R stands for REAL, I for INTEGER, D for DOUBLE PRECISION and V for VARIABLE. So MVV will be the suffix of a routine that implements (VARIABLE) * (VARIABLE) and DDV of a routine that implements (DOUBLE PRECISION)/(VARIABLE).

The suffix of the LOGICAL functions implementing the relational operators is composed of the two letters representing the operator and the letter V. So .LT. is implemented by a function with suffix LTV. The suffix of the routine implementing the norm function is NRM, Error routine - ERR and copy routine - CPY.

2.4. Using the Package with AUGMENT.

Writing the Source Code:

In order to get derivatives of a function, say the Taylor series expansion of a function $\hat{f}$, the user should write an "extended" FORTRAN function or subroutine that computes $\hat{f}$. All legal FORTRAN constructs can be used. All program variables which depend on the independent variable should be declared as type TAYLOR, including the function itself. Program variables which do not depend on the independent variable can be of any other type.

External functions and subroutines can be used as part of the computation. Functions must be declared as type TAYLOR. External functions and subroutines can be translated separately. However, one has to make sure that the number of computer words reserved for each TAYLOR variable in the external subroutine (function) is the same as the number of words reserved in the calling routine.

The Description Deck:

The next step is to submit the source deck with the description deck as data to AUGMENT. See Appendix C for the deck structure. The description deck is supplied with the package.

However, since the number of words reserved for each VARIABLE changes from problem to problem, this number has to be put into the description deck each time. To make it easier, the description deck was split into two parts: HEAD and BODY. The number of words to reserve is inserted in between the two parts. This number should be an integer constant. Column 1 in the card holding this number must be left blank.

Order of Differentiation:

The routines in both packages were designed to implement any "order" of differentiation without the need to be recompiled every time the "order" is changed. The "order" of differentiation is provided to the package through a common block. In TAYLOR by COMMON/DEGREE/N and in GRADIENT by COMMON/ORDER/N N is the order of the operator, that is: if N = 1 the package will compute function values only; if N = 2, function values and first derivatives (or first partial with respect to one variable); and so on. The routines in the package do not check that there is enough space provided for the VARIABLES. However there is a check that $N \geq 1$. The order can be changed at run time but care should be taken not to exceed the number of words provided for each VARIABLE.

Working Space:

Some routines in TAYLOR need work space and, since they are designed to handle any order of differentiation, the work space has to be provided by the user. The work space is provided through four common blocks.

```
COMMON/WORK1/WORK1(N)
COMMØN/WORK2/WORK2(N)
COMMØN/WORK3/WORK3(N)
COMMON/WORK4/WORK4(N)
```

N should be the highest order of differentiation used. The GRADIENT package does not require work space.

Using the Translated Routine:

The translated routine is a FORTRAN subroutine that gets as input the value of its arguments and their derivatives, and gives as output the value of the function and its derivatives. For example, in the Taylor package, if t is the independent variable, then $t, 1, 0, 0 \ldots$ is the Taylor series expansion of t. In the Gradient package, if x is the j^{th} independent variable then $\frac{\partial x}{\partial x_j} = 1$ and $\frac{\partial x}{\partial x_i} = 0$ for $i \neq j$.

Example:

In this example we compute the first 9 terms of the Taylor series expansion of $f(t) = \exp(\cos(4t)) + \arctan(\sin^2(t))$ at the point $t = .5$.

a) The source code:

```
      TAYLOR FUNCTION FUN(T)
      TAYLOR T
      FUN=EXP(COS(4.*T))+ATAN(SIN(T)**2)
      RETURN
      END
```

b) The translated code:

```
      SUBROUTINE FUN (T, TYLRLT)
C                ===== PROCESSED BY AUGMENT, VERSION 4L =====
C                ----- TEMPORARY STORAGE LOCATIONS -----
C                      TAYLOR
      REAL TYLTMP(9,2)
C                ----- LOCAL VARIABLES -----
C                      TAYLOR
      REAL TYLRES(9)
C                ----- GLOBAL VARIABLES -----
C                      TAYLOR
      REAL T(9), TYLRLT(9)
C                ===== TRANSLATED PROGRAM =====
      CALL TYLMRV (4.,T,TYLTMP(1,1))
      CALL TYLCOS (TYLTMP(1,1),TYLTMP(1,1))
      CALL TYLEXP (TYLTMP(1,1),TYLTMP(1,1))
      CALL TYLSIN (T,TYLTMP(1,2))
      CALL TYLEVI (TYLTMP(1,2),2,TYLTMP(1,2))
      CALL TYLATN (TYLTMP(1,2),TYLTMP(1,2))
      CALL TYLAVV (TYLTMP(1,1),TYLTMP(1,2),TYLRES)
      GO TO 30000
C                ----- RETURN CODE -----
30000 CONTINUE
      CALL TYLCPY (TYLRES,TYLRLT)
      RETURN
       END
```

c) Main program:

```
      DIMENSION T(9),FUNC(9)
      COMMON /DEGREE/ N
      COMMON /WORK1/W1(9)
      COMMON /WORK2/W2(9)
      COMMON /WORK3/W3(9)
      COMMON /WORK4/W4(9)
      N=9
      T(1)=.5
      T(2)=1.
      DO 10 I=3,9
10    T(I)=0.
      CALL FUN(T,FUNC)
      PRINT91,T(1),(FUNC(L),L=1,9)
91    FORMAT(1X,'EXAMPLE: TAYLOR SERIES EXPANSION',/,3X,'T=',F6.3//,
     *  (3X,E13.5))
      STOP
      END
```

d) Output:

```
 EXAMPLE: TAYLOR SERIES EXPANSION
   T=  .500

       .88551+00
      -.15998+01
       .69251+01
      -.77435+01
      -.34296+01
       .35406+02
      -.59899+02
       .28534+02
       .10762+03
```

In Appendix B we give a more complicated example.

Error Handling:

In general the packages do not check that the arguments are within the domain of definition of the functions. Checking is done only for division by REAL, INTEGER or VARIABLE types. The packages also provide the capability to specify what constitutes division by zero. If the absolute value of an argument to division routine is smaller than or equal to specified value, error occurs. This value is set initially to 0.0 by a DATA statement. However it can be changed in runtime through common block

/ZERO/EPS. The routines in the package also check that the order of differentiation is at least 1. In case an error is discovered, the error routine is called (suffix ERR). The error routine prints error messages and performs a "walk back" (which traces the sequence of subprogram calls back to the main program) and stops.

Non ANSI FORTRAN Parts:

No special care was taken to comply with some of the restrictions imposed by ANSI FORTRAN, for two reasons:

a) Some of the features in the packages could not have been implemented otherwise.

b) Some of the restrictions violated seem to us arbitrary and unreasonable.

Moreover they do not exist in most production compilers.

Below we give the list of non ANSI FORTRAN constructions used.

1) The set of "standard" functions used is larger than the set in ANSI FORTRAN. Functions which are not in the Standard are flagged in the function table by †. The corresponding routines could be deleted or modified to fit different systems.

2) The work space to TAYLOR is provided through common blocks and therefore the common block sizes in the TAYLOR routines would be different from the block sizes in the main program.

3) The walk back routine used in the error handling routine is non-standard and has to be changed in other systems.

4) The REAL variable EPS appears in common block /ZERO/ and in DATA statement (in the error routine).

5) No care was taken to comply with the standard concerning expressions as subscripts to arrays.

6) The function ATAN2 is not implemented.

7) No care was taken not to mix INTEGER with REAL.

Using the Package; Summary:

Assume one has all the package subroutines in relocatable form, so they can be used as library routines. In order to get the derivatives of a function $\hat{f}$ one has to do the following:

A) Write a FORTRAN subroutine[†] (function) that computes the function $\hat{f}$. In the subroutine, declare all FORTRAN variables that depend on the independent variables as type TAYLOR (or GRADIENT). The rest of the variables can be of any other type.

B) Insert the number of computer words reserved for each VARIABLE into the description deck.

C) Submit the source deck with the appropriate description deck as data to AUGMENT (See Appendix C and also [3]).

D) Submit the output from AUGMENT as data to the FORTRAN compiler.

E) Call the subroutine with the desired arguments.

[†]One can use main programs too but that is a more complicated way of doing it.

Remarks

1) Always remember that the initial values of the derivatives of the input variables have to be provided too.

2) All the restrictions that apply to the use of AUGMENT apply to the packages.

3) AUGMENT does not translate I/O and DATA statements. Their translation has to be done by hand.

4) This report is by no means a substitute for AUGMENT user information manual (MRC TSR #1469 [3]). The user should be familiar with that report.

Acknowledgments

The author is indebted to Dr. C. de Boor and Dr. S. V. Parter for many stimulating discussions, constructive criticism and valuable suggestions. The author wishes to thank Dr. F. Crary and Dr. L. Landweber for critical reading of the manuscript and for many valuable suggestions.

References

1. Barton, D., Willer, I. M. and Zahar, R. V. M., Taylor Series Method for Ordinary Differential Equations. An Evaluation. Mathematical Software, 1971, Academic Press, Inc., New York and London.

2. Braun, J. A. and Moore, R. E., A program for the solution of differential equations using Interval Arithmetic (DIFEQ) for the CDC3600 and 1604. MRC Technical Summary Report #901, June 1968.

3. Crary, F. D., The AUGMENT Precompiler : I. User Information. MRC Technical Summary Report #1469, December 1974.

4. Crary, F. D., The AUGMENT Precompiler: II. Technical Documentation, MRC Technical Summary Report #1420, October 1975.

5. Knapp, H. and Wanner, G. LIESE II. A Program for Ordinary Differential Equations using Lie-Series, MRC Technical Summary Report #1008, March 1970.

6. Krasnosel'skii, M. A. et. al., Approximate Solution of Operator Equations, Wolters-Noordhoff, Groningen, 1972.

7. Moore, R. E., The Automatic Analysis and Control of Error in Error in Digital Computation, Vol. 1, Editor L. B. Rall, John Wiley and Sons, Inc. New York 1965.

8. Moore, R. E., Interval Analysis, Prentice Hall 1966.

9. Pugh, R. E., A language for nonlinear programming problems, Mathematical Programming 2 (2) 176-206, 1972.

10. Reiter, A. Automatic generation of Taylor coefficients (TAYLOR) for the CDC 1604, MRC Technical Summary Report #830, November 1967.

11. Reiter, A, and Gray J., Compiler of Differentiable Expressions (CODEX) for the CDC 3600, MRC Technical Summary Report #791, December 1967.

12. Wertz, H. J., SUPER-CODEX: Analytic Differentiation of FORTRAN Statements. Aerospace Corporation, Los Angeles, California. Aerospace report No TOR-0172(9320)-12, June 1972.

13. Kuba, D. and Rall, L. B., A UNIVAC 1108 program for obtaining rigorous error estimates for approximate solution of systems of equations, MRC Technical Summary Report #1168, January 1972.

14. Gray, J. and L. B. Rall, NEWTON: A general purpose program for solving nonlinear systems. MRC Technical Summary Report #790, September 1967.

15. W. Rudin, Real and Complex Analysis. McGraw-Hill, 1966.

Appendix A

Recurrence Relations for Taylor Series Expansion:

a) $Z = X \pm Y$

$$[Z]_J = [X]_J \pm [Y]_J \qquad J = 0,1,\ldots$$

b) $Z = X \cdot Y$

$$[Z]_J = \sum_{k=0}^{J} [X]_k \cdot [Y]_{J-k} \qquad J = 0,1,\ldots$$

c) $Z = X/Y$

$$[Z]_J = 1/Y \{[X]_J - \sum_{k=0}^{J-1} [Z]_k \cdot [Y]_{J-k}\} \qquad J = 0,1,\ldots$$

d) $Z = X^I$ $\qquad$ I integer

1) $I > 0$, $\quad I = \sum_{S=0}^{n} L_S 2^S \qquad L_S \in \{0,1\}$

$$[Z]_J = [\prod_{S=0}^{n} X^{L_S \cdot 2^S}]_J \qquad J = 0,1,\ldots$$

2) $I = 0 \; ; \; Z \equiv 1$

3) $I < 0 \qquad [Z]_J = [1/X^{-I}]_J \qquad J = 0,1,\ldots$

e) $Z = X^a$ $\qquad$ a real constant

$$[Z]_J = 1/X \cdot \sum_{k=0}^{J-1} ((a(J-k)-k)/J) \cdot [Z]_k \cdot [X]_{J-k} \qquad J = 1,2,\ldots$$

f) $Z = X^Y$

$$[Z]_J = [EXP(Y \cdot LOG(X))]_J \qquad j = 0,1,\ldots$$

g) $Z = LOG(X)$

$$[Z]_1 = [X]_1/X$$

$$[Z]_J = 1/X \{[X]_J - \sum_{k=0}^{J-1} ((J-k)/J)[X]_{J-k} \cdot [Z]_k \qquad J = 2,3,\ldots$$

h) $Z = EXP(X)$

$$[Z]_J = \sum_{k=0}^{J-1} ((J-k)/J)[Z]_k \cdot [X]_{J-k} \qquad J = 1,2,\ldots$$

i) $Z = \mathrm{SIN}(X)$, $W = \mathrm{COS}(X)$

$$[Z]_J = \sum_{k=0}^{J-1} ((J-k)/J)[W]_k \cdot [X]_{J-k} \qquad J = 1,2,\ldots$$

$$[W]_J = -\sum_{k=0}^{J-1} ((J-k)/J)[Z]_k \cdot [X]_{J-k}$$

j) $Z = \mathrm{SINH}(X)$, $W = \mathrm{COSH}(X)$

$$[Z]_J = \sum_{k=0}^{J-1} ((J-k)/J)[W]_k \cdot [X]_{J-k}$$

$$J = 1,2,\ldots$$

$$[W]_J = \sum_{k=0}^{J-1} ((J-k)/J) \cdot [Z]_k \cdot [X]_{J-k}$$

k) $Z = \mathrm{ATAN}(X)$

Let $V = X^2 + 1$ $W = 1/V$

$$[Z]_J = \sum ((J-k)/J)\, [W]_k \cdot [X]_{J-k}$$

ℓ) $Z = \mathrm{ASIN}(X)$, $Y = \mathrm{ACOS}(X)$

Let $V = 1 - X^2$ $W = 1/\sqrt{V}$

$$[Z]_J = \sum_{k=0}^{J-1} ((J-k)/J)[W]_k \cdot [X]_{J-k} \qquad J = 1,2,\ldots$$

$$[Y]_J = -\sum_{k=0}^{J-1} ((J-k)/J)\, [W]_k \cdot [X]_{J-k} \qquad J = 1,2,\ldots$$

m) $\mathrm{TAN}(X) = \mathrm{SIN}(X)/\mathrm{COS}(X)$

$\sqrt{x} = X^{\frac{1}{2}}$

$\mathrm{TANH}(X) = \mathrm{SINH}(X)/\mathrm{COSH}(X)$

Appendix B

Example

The following example was taken from a homework assignment given in an optimization class at the University of Wisconsin. This example is simple but quite typical of problems that arise in applications. We have to compute the gradient of a function which can be easily described by a computer program, but whose explicit expression is quite hard to obtain. When one tries to compute the gradient numerically, one runs into convergence problems. In this example we show how easy it is to get the gradient of such a function by using the GRADIENT package.

The Problem

We have a missile on the north pole of a ball of radius 1 and we want to fly to the south pole. (The units are chosen in such a way that all the constants are 1). Because the problem is symmetric we only have to solve a two dimensional problem.

The motion equations are:

$$\frac{d^2r}{dt^2} = -\frac{r}{\|r\|^3} + u \qquad r = (x,y)$$

Let $t = \alpha\tau$ then

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ \xi \\ \eta \end{pmatrix} = \alpha\left[\begin{pmatrix} \xi \\ \eta \\ \dfrac{-x}{(x^2+y^2)^{2/3}} \\ \dfrac{-y}{(x^2+y^2)^{3/2}} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ u_1 \\ u_2 \end{pmatrix}\right] = \alpha f(x,y,\xi,\eta,u_1,u_2) .$$

Discretize by the Euler method

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ \xi_{k+1} \\ \eta_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \\ \xi_k \\ \eta_k \end{pmatrix} + h\alpha f(x_k, y_k, \xi_k, \eta_k, u_{1,k}, u_{2,k})$$

$k = 0,1,\ldots,g \qquad h = 0,1\,.$

Finally, solve:

$$\min h(\alpha, u_{1,0}, u_{2,0}, u_{1,8}, u_{2,8}, u_{1,9}, u_{2,9})\,.$$

$$= \{20[(x_{10})^2 + (y_{10}+1)^2 + (\xi_{10})^2 + (\eta_{10})^2] + (u_{1,0}^2)^2 + (u_{2,0})^2$$

$$+ (u_{1,8})^2 + (u_{2,8})^2 + (u_{1,9})^2 + (u_{2,9})^2 + \frac{\alpha^2}{10} + \frac{20}{x_5^2 + y_5^2}\}\,.$$

All $u_{1,j} \equiv 0 \quad u_{2,j} \equiv 0 \quad$ for $\quad 1 \le j < 8$.

Use the variable metric algorithm to solve the above minimization problem.

1. INPUT DECK

```
@XQT MRC*LIB.AUGMENT
@ADD GK*DIFF.DESC-GRD/HEAD
 8
@ADD GK*DIFF.DESC-GRD/BODY
*BEGIN
:FOR,IS TEST1
       IMPLICIT GRADIENT (A-H,O-Z)
       GRADIENT FUNCTION FLTFN(ALFA,U0,U8,U9)
       DIMENSION U0(2), U8(2),U9(2),U(2,10), X(11),Y(11),VY(11),VX(11)
       X(1)=0.0
       Y(1)=1.0
       VX(1)=0.0
       VY(1)=0.0
       H=.1
       DO10I=1,10
       U(1,I)=0.0
       U(2,I)=0.0
10     CONTINUE
       U(1,1)=U0(1)
       U(2,1)=U0(2)
       U(1,9)=U8(1)
       U(2,9)=U8(2)
       U(1,10)=U9(1)
       U(2,10)=U9(2)
       DO20 I=1,10
       RS=(X(I)**2+Y(I)**2)**1.5
       X(I+1)=X(I)+H*ALFA*VX(I)
       Y(I+1)=Y(I)+H*ALFA*VY(I)
       VX(I+1)=VX(I)+H*ALFA*(U(1,I)-X(I)/RS)
       VY(I+1)=VY(I)+H*ALFA*(U(2,I)-Y(I)/RS)
20     CONTINUE
       FLTFN=20.*(X(11)**2+(Y(11)+1)**2+VX(11)**2+VY(11)**2)
      $  + U0(1)**2+U0(2)**2+U8(1)**2+U8(2)**2+U9(1)**2+U9(2)**2
      $  +(ALFA**2)/10.0 +20.0/(X(6)**2+Y(6)**2)
       RETURN
       END
*END
```

Remarks.

1) @XQT MRC*LIB.AUGMENT starts the execution of AUGMENT.

2) @ADD GK*DIFF.DESC-GRD/HEAD, adds the card image of the HEAD part of the description deck into the run stream.

Similarly @ADD GK*DIFF.DESC-GRD/BODY Adds the BODY part.

3) The function is translated into a subroutine. The function and gradient values are stored in the last argument of the subroutine.

4) The translated subroutine can be used to compute function and gradient values or function values alone. (See Order of Differentiation)

5) The rest of the computation details are omitted.

OUTPUT FROM AUGMENT

```
      SUBROUTINE FLTFN (ALFA,U0,U8,U9, GRDRLT)
C               ===== PROCESSED BY AUGMENT, VERSION 4H =====
C               ----- TEMPORARY STORAGE LOCATIONS -----
C                     GRADIENT
      REAL GRDTMP(8,3)
C               ----- LOCAL VARIABLES -----
      INTEGER I
C                     GRADIENT
      REAL H(8), RS(8), U(8,2,10), VX(8,11), VY(8,11), X(8,11), Y(8,11),
     *     GRDRES(8)
C               ----- GLOBAL VARIABLES -----
C                     GRADIENT
      REAL ALFA(8), U0(8,2), U8(8,2), U9(8,2), GRDRLT(8)
C               ===== TRANSLATED PROGRAM =====
      CALL GRDCFR (0.0,X(1,1))
      CALL GRDCFR (1.0,Y(1,1))
      CALL GRDCFR (0.0,VX(1,1))
      CALL GRDCFR (0.0,VY(1,1))
      CALL GRDCFR (.1,H)
      DO 10     I=1,10
      CALL GRDCFR (0.0,U(1,1,I))
      CALL GRDCFR (0.0,U(1,2,I))
10    CONTINUE
      CALL GRDCPY (U0(1,1),U(1,1,1))
      CALL GRDCPY (U0(1,2),U(1,2,1))
      CALL GRDCPY (U8(1,1),U(1,1,9))
      CALL GRDCPY (U8(1,2),U(1,2,9))
      CALL GRDCPY (U9(1,1),U(1,1,10))
      CALL GRDCPY (U9(1,2),U(1,2,10))
      DO 20     I=1,10
      CALL GRDEVI (X(1,I),2,GRDTMP(1,1))
      CALL GRDEVI (Y(1,I),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVR (GRDTMP(1,2),-1.5,RS)
      CALL GRDMVV (H,ALFA,GRDTMP(1,1))
      CALL GRDMVV (GRDTMP(1,1),VX(1,I),GRDTMP(1,1))
      CALL GRDAVV (X(1,I),GRDTMP(1,1),X(1,I+1))
      CALL GRDMVV (H,ALFA,GRDTMP(1,1))
      CALL GRDMVV (GRDTMP(1,1),VY(1,I),GRDTMP(1,1))
      CALL GRDAVV (Y(1,I),GRDTMP(1,1),Y(1,I+1))
      CALL GRDMVV (H,ALFA,GRDTMP(1,1))
      CALL GRDDVV (X(1,I),RS,GRDTMP(1,2))
      CALL GRDSVV (U(1,1,I),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDMVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDAVV (VX(1,I),GRDTMP(1,2),VX(1,I+1))
      CALL GRDMVV (H,ALFA,GRDTMP(1,1))
      CALL GRDDVV (Y(1,I),RS,GRDTMP(1,2))
      CALL GRDSVV (U(1,2,I),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDMVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDAVV (VY(1,I),GRDTMP(1,2),VY(1,I+1))
```

```
20    CONTINUE
      CALL GRDEVI (X(1,11),2,GRDTMP(1,1))
      CALL GRDAVI (Y(1,11),1,GRDTMP(1,2))
      CALL GRDEVI (GRDTMP(1,2),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVI (VX(1,11),2,GRDTMP(1,1))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,1),GRDTMP(1,1))
      CALL GRDEVI (VY(1,11),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDMRV (20.,GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVI (U0(1,1),2,GRDTMP(1,1))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,1),GRDTMP(1,1))
      CALL GRDEVI (U0(1,2),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVI (U8(1,1),2,GRDTMP(1,1))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,1),GRDTMP(1,1))
      CALL GRDEVI (U8(1,2),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVI (U9(1,1),2,GRDTMP(1,1))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,1),GRDTMP(1,1))
      CALL GRDEVI (U9(1,2),2,GRDTMP(1,2))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,2),GRDTMP(1,2))
      CALL GRDEVI (ALFA,2,GRDTMP(1,1))
      CALL GRDDVR (GRDTMP(1,1),10.0,GRDTMP(1,1))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,1),GRDTMP(1,1))
      CALL GRDEVI (X(1,6),2,GRDTMP(1,2))
      CALL GRDEVI (Y(1,6),2,GRDTMP(1,3))
      CALL GRDAVV (GRDTMP(1,2),GRDTMP(1,3),GRDTMP(1,3))
      CALL GRDDRV (20.0,GRDTMP(1,3),GRDTMP(1,3))
      CALL GRDAVV (GRDTMP(1,1),GRDTMP(1,3),GRDRES)
      GO TO 30000
C                ----- RETURN CODE -----
30000 CONTINUE
      CALL GRDCPY (GRDRES,GRDRLT)
      RETURN
      END
```

Appendix C[†]

Deck Arrangement

The data deck read by AUGMENT has the structure shown in the following diagram:

```
                  -----------------------------------------------
                 /*END                                          |
                 |                                              |
                                                                |
             --------------------------------------------------
            /                  Source Deck                   /|
           /---------------------------------------------- / |
          /                                               |  |
          |                                               |
                                                          |
        ---------------------------------------------------
       /*BEGIN                                            |
       |                                                  |
                                                          |
     -----------------------------------------------------
    /               Description Deck                    /|
   /--------------------------------------------------/ |
  /                                                   |  |
  |                                                   |
  |                                                   |
```

At the conclusion of processing, the translated program decks are in the output file in 80 column card image format.

[†]This page is taken out of: The AUGMENT Precompiler 1. User Information. Fred Crary [3].

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 1697 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS. ✓		5. TYPE OF REPORT & PERIOD COVERED Summary Report - no specific reporting period
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) G./Kedem		8. CONTRACT OR GRANT NUMBER(s) (15) DAAG29-75-C-0024 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of Wisconsin, 610 Walnut Street, Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709 (11)		12. REPORT DATE November 1976
		13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (9) Technical summary rept.,		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(12) 55p. (14) MRC-TSR-1697

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Factorable functions

Automatic differentiation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A method for the automatic differentiation of computer functions (subroutines) written in a high level language is discussed.

A theory is developed to show that most functions that arise in applications can be differentiated automatically. It is shown how one can take a FORTRAN function (subroutine) and, with the aid of a precompiler, obtain a FORTRAN subroutine that computes the original function and its desired derivatives.

Implementation of two types of differentiation is described:

1) Automatic Taylor series expansion of FORTRAN programs.
2) Automatic Gradient calculation of FORTRAN functions.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 221 200 LB

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 1697 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS. ✓		5. TYPE OF REPORT & PERIOD COVERED Summary Report - no specific reporting period
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 G. Kedem		8. CONTRACT OR GRANT NUMBER(s) 15 DAAG29-75-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of 610 Walnut Street Wisconsin Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P.O. Box 12211 Research Triangle Park, North Carolina 27709		12. REPORT DATE 11 November 1976
		13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 9 Technical summary rept.,		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. 14 MRC-TSR-1697

12 55p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Factorable functions

Automatic differentiation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A method for the automatic differentiation of computer functions (subroutines) written in a high level language is discussed.

A theory is developed to show that most functions that arise in applications can be differentiated automatically. It is shown how one can take a FORTRAN function (subroutine) and, with the aid of a precompiler, obtain a FORTRAN subroutine that computes the original function and its desired derivatives.

Implementation of two types of differentiation is described:
1) Automatic Taylor series expansion of FORTRAN programs.
2) Automatic Gradient calculation of FORTRAN functions.